A Scalable Decision Procedure for Fixed-Width Bit-Vectors

Roberto Bruttomesso
Università della Svizzera Italiana
Lugano, Switzerland
roberto.bruttomesso@usi.ch

Natasha Sharygina
Università della Svizzera Italiana
Lugano, Switzerland
natasha.sharygina@usi.ch

ABSTRACT

Efficient decision procedures for bit-vectors are essential for modern verification frameworks. This paper describes a new decision procedure for the core theory of bit-vectors that exploits a reduction to equality reasoning. The procedure is embedded in a congruence closure algorithm, whose data structures are extended in order to efficiently manage the relations between bit-vector slicings, modulo equivalence classes. The resulting procedure is incremental, backtrackable, and proof producing: it can be used as a theory-solver for a lazy SMT schema. Experiments show that our approach is comparable and often superior to bit-blasting on the core fragment, and that it also helps as a theory layer when applied over the full bit-vector theory.

1. INTRODUCTION

Fixed-width bit-vectors are fundamental data structures commonly used to model hardware components, like registers and memories, as well software constructs, such as basic data-types. Properties on bit-vectors can be expressed by means of a wide range of operators and relations, whose semantics is defined in first order theory $\mathcal{BV}$, the theory of bit-vectors.

State-of-the-art decision procedures for $\mathcal{BV}$ fall into two extremes. On the one hand, the system is decomposed into a bit-level representation, and encoded into Boolean logic (*bit-blasting*): each bit-vector can be encoded with a set of Boolean variables, in the worst case one for each bit, while operators and properties are encoded into propositional formulæ. A SAT-Solver can then be employed as a decision procedure. The major drawback of bit-blasting is that the encoding of bit-vectors in terms of unrelated objects (bits) results in the loss of the structure of the original design. On the other hand some verifiers resort to decision procedures that reason over bit-vectors as a whole, by keeping them as individual entities, for instance by encoding words into integers. This approach seems to be useful when large datapaths are considered. However it may become inefficient when the Boolean component is predominant.

An emerging technology is SMT (Satisfiability Modulo Theories), a new generation of theorem provers that combine the efficiency of modern SAT-Solvers and theory-specific decision procedures (*theory-solvers*). Most SMT-Solvers follow the so-called *lazy* approach where the theory-solver is called on demand to detect the satisfiability of a set of constraints enumerated by a DPLL SAT-Solver. The lazy approach was shown to be effective for SMT.

In this paper we present a new decision procedure for the *core* fragment of $\mathcal{BV}$ (for equalities, extraction, and concatenation, $\mathcal{BV}_C$) by developing a novel and efficient theory-solver for a lazy SMT schema. The core fragment, although decidable in polynomial time, is often handled by bit-blasting (thus being solved as if it were an NP-Complete problem). Our approach, instead, is inspired by the work of [8, 4], and it relies on a reduction to the theory of equality ($\mathcal{E}$) in order to preserve the word-level structure of the problem and to use decision procedures that run in polynomial time.

In summary, the contribution of this paper is an incremental, backtrackable and proof-producing decision procedure for the core fragment, based on a state-of-the-art congruence closure algorithm, that can be integrated in the lazy schema for SMT. In particular:

- We investigate the theoretical aspects of an incremental reduction of bit-vector equalities into $\mathcal{E}$ by introducing the notion of *base*, an elegant and extensible representation for bit-vector decompositions that guarantees the completeness of the approach.
- We define the concept of *coarsest base*, the decomposition for bit-vectors that minimizes the amount of slicing on the fly, by exploiting a hierarchical structure of equivalence classes.
- We show how our approach can be efficiently implemented in a classical congruence closure algorithm with the help of a *novel data-structure*, the CBE (*coarsest base modulo equivalence*), that represents decompositions and equivalence classes in a simple and compact manner. The algorithm is incremental, backtrackable, and it partially handles uninterpreted functions and negated equalities.
- We empirically demonstrate the applicability and the advantages of our method, implemented in OpenSMT [19], through a set of experiments.

The rest of the paper is structured as follows. In §2 we recall some background notions. §3 describes our reduction

from $\mathcal{BV}_C$ to $\mathcal{E}$, that can be efficiently implemented using the notion of CBE, as shown in §4. §5 reports on the applicability of the solver for SMT. In §6 we evaluate our approach with respect to bit-blasting. We conclude in §7.

1.1 Related Work

Cyrluk et al. [8] present a canonizer and a solver for $\mathcal{BV}_C$ based on a union-find algorithm (we discuss this approach in more detail in §2.4). In contrast to our approach their procedure is neither incremental nor backtrackable; no particular insight is given about data structures to be used in a potential implementation and no experiments are reported. In addition our procedure can partially handle negated equalities and uninterpreted functions.

A noteworthy extension to [8] is given in [4], where the basic method is refined in particular to take into account the case of bitwise operators. The algorithm is defined to be integrated in Shostak's combination framework, and it consists of a set of normalization functions for bit-vectors and slicing functions (cut, slice, dice) that take care of bit-vectors decompositions. [4] also presents an extension to non-fixed-width bit-vectors. Our approach also uses slicing functions but, in contrast, it avoids duplicated work, in particular when new equalities are asserted, by exploiting equivalence classes. Moreover, our procedure is designed for the integration in SMT-Solvers: it is defined as an extended version of a congruence closure algorithm, and it acts as a theory-layer for the core fragment, thus improving the subsequent complete reduction to SAT.

[12] presents a decision procedure that handles separately arithmetic operators and bitwise plus core operators, in two different theory-solvers. Each solver iteratively computes a normal form for bit-vectors (arithmetic normal form and concatenational normal form) by means of a set of rewrite rules. Information is exchanged by means of a Nelson-Oppen-like communication. Our approach, instead, is based on a congruence closure algorithm for the core fragment only, and it is used as a layer before resorting to bit-blasting.

MathSAT [5] is a lazy SMT-Solver, and it employs a layered approach, composed by a solver for $\mathcal{EUF}$, and an engine based on a set of inference rules. In contrast to our method, the intermediate layers of MathSAT are not complete w.r.t. $\mathcal{BV}_C$. To the best of our knowledge, tools Z3 [9], Yices [11] and CVC3 [1] implement a lazy SMT schema that resort to bit-blasting after a layer for $\mathcal{EUF}$, without any specific reasoning on $\mathcal{BV}_C$. STP [13] relies on a polytime decision procedure [2] for arithmetic before resorting to bit-blasting. Other tools for $\mathcal{BV}$ based on eager reductions to SAT are BAT [16], Beaver [14], Boolector [3], Spear [21], Sword [23], Uclid [7].

2. BACKGROUND

2.1 Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) is the problem of deciding the satisfiability of a (often quantifier-free) first order formula with respect to a background decidable theory $\mathcal{T}$ (SMT($\mathcal{T}$)). An SMT($\mathcal{T}$) formula is an arbitrary Boolean combination of predicates of $\mathcal{T}$ (*theory-atoms*).

Theories of interest are the ones of equality ($\mathcal{E}$), uninterpreted functions and predicates ($\mathcal{EUF}$), linear arithmetic over the reals or the integers ($\mathcal{LRA}$, $\mathcal{LIA}$), arrays ($\mathcal{A}$), and bit-vectors ($\mathcal{BV}$). The following is an instance of an SMT($\mathcal{LRA}$) formula:

$$((x - y < 10) \wedge (x + y = 5)) \vee (y < 0)$$

The *lazy approach* is a well-established schema used in many state-of-the-art SMT-Solvers such as z3, Yices, MathSAT, Barcelogic, and CVC3; it is based on a tight cooperation between a DPLL SAT-Solver, used as a model enumerator, and a decision procedure for $\mathcal{T}$. For a complete survey on lazy-SMT we refer the reader to [20].

2.2 A theory of Bit-Vectors

A bit-vector is an array of individual bits. In this work we restrict our attention to *fixed width* bit-vectors. We use the notation $p_{[n]}$ to indicate a bit-vector term of width n, or simply p when the width is not important or can be deduced from the context. We use letters x, y, z to denote variables, p, q, s, t, u for generic terms, n, m for bit-vectors width, and i, j, k, l for extraction indexes. A constant is either denoted with c, or explicitly as a string of 0s and 1s, as for instance 0010. For all bit-vectors the most significant bit is the left-most bit.

In the theory of bit-vectors ($\mathcal{BV}$) terms can be built from variables and constants with the application of a set of *operators*: *Extraction* $x_{[n]}[i:j]$ $(n-1 \geq i \geq j \geq 0)$: represents a bit-vector of width $(i-j+1)$ whose k-th bit is equivalent to the $k+j$-th bit in x. *Concatenation* $x_{[n]} :: y_{[m]}$: represents a bit-vector of width $(n+m)$ whose k-th bit is equivalent to the k-th bit of y, if $k < m$, and to the $(k+m)$-th bit of x otherwise. *Arithmetic* operators: include usual arithmetic operators, such as addition $x_{[n]} + y_{[n]}$, multiplication $x_{[n]} \cdot y_{[n]}$, shifts $x_{[n]} \ll k$, etc. *Bitwise* operators: include logical operators like *and* $x_{[n]} \& y_{[n]}$, *not* $\overline{x}_{[n]}$ that compute standard logical operations among corresponding bits of its arguments.

Predicates such as *equality* $=$ or *less than* $<$ can be used to express relations between bit-vector terms (we refer the reader to [6] for a thorough presentation of bit-vector syntax and semantics). The decision problem for a quantifier-free *conjunction* of bit-vector atoms in $\mathcal{BV}$ is known to be NP-Complete. However, the complexity reduces to P if only extraction, concatenation and positive equalities are considered [8]. We shall refer to this sub-theory with $\mathcal{BV}_C$ (*core* theory of bit-vectors).

Core operators are commonly employed to encode the data-path in a circuit, when data is extracted from a register or when multiple wires combine into a single bus. They are supported in widely-used hardware description languages such as Verilog and VHDL. Moreover other operators can be rephrased in terms of extraction and concatenation. We report few examples:

- bit-masks: $x_{[8]}$ & 00001111 $= 0000 :: x[3:0]$
- shifts: $x_{[8]} \ll 2 = x[5:0] :: 00$
- multiplication by a power of two: $x_{[8]} \cdot 00000010 = x[6:0] :: 0$
- zero/sign extensions: $ZE(8, x_{[6]}) = 00 :: x$, $SE(8, x_{[7]}) = x[6:6] :: x$

2.3 Union-find algorithms and extensions

A *union-find* algorithm [22] is a decision procedure for $\mathcal{E}$. It receives as input a set of equalities E between elements

in a set $\mathcal{X}$ which may contain variables and constants. A union-find algorithm maintains the collection of *equivalence classes* induced by the axioms of equality and E. The algorithm works incrementally. Initially each element is associated with an individual equivalence class. Each class can be efficiently maintained by keeping a *representant*. The two main operations are *union* and *find*, that merge two equivalence classes and retrieve the representant of the class respectively. An inconsistency is detected when trying to merge two classes containing different constants.

A widely used variation of [22] is the so-called *quick-find* approach of [15, 10], where representants can be accessed in constant time from any member of the equivalence class. Most union-find algorithms rely on a graph-based representation, where the nodes are the members of the equivalence classes, and the edges connect elements in the same class. In the quick-find approach, a node can be stored with a record `node`, containing a `rep` field, of type `(node *)` (pointer to `node`) that points to the representant of the class. From now the node associated with an element x will be written using the `typewriter` font as `x`. Merging two equivalence classes A and B, with $|A| < |B|$, corresponds to updating the `rep` pointers of the nodes of A to point to the node corresponding to the representant of B.

In the rest of the paper we shall refer to the union-find algorithm with *AssertEq*.

2.4 The approach of Cyrluk et al.

Cyrluk et al. [8] propose a method to check the satisfiability of a set E of $\mathcal{BV}_C$-equalities by reducing E into an equisatisfiable set of equalities E' over $\mathcal{E}$. The reduction works as follows. Each variable and constant appearing in a $\mathcal{BV}_C$-equality $p = q$ is decomposed into the concatenation of individual bits as $p(n-1) :: \ldots :: p(0) = q(n-1) :: \ldots :: q(0)$; E' is constructed with the set of equalities $p(i) = q(i)$, for each bit-vector position i. The original set of $\mathcal{BV}_C$-equalities E is unsatisfiable if and only if E' is unsatisfiable over $\mathcal{E}$. In Example 1 E' (and therefore E) is unsatisfiable for its subset $\{x(3) = 0, x(3) = y(3), y(3) = 1\}$. E' can is checked for consistency by using a union-find algorithm.

EXAMPLE 1. *Consider the following set of $\mathcal{BV}_C$-equalities over two variables $x_{[8]}, y_{[8]}$*

$$E = \{x[5:0] = 010110, y[7:2] = 000110, x = y\}$$

Each equality in E is decomposed, resulting in the set E'

$$E' = \left\{ \begin{array}{c} x(5) = 0,\ x(4) = 1,\ x(3) = 0,\ x(2) = 1 \\ x(1) = 1,\ x(0) = 0,\ y(7) = 0,\ y(6) = 0 \\ y(5) = 0,\ y(4) = 1,\ y(3) = 1,\ y(2) = 0 \\ x(7) = y(7),\ \ldots,\ x(0) = y(0) \end{array} \right\}$$

E' can be proven inconsistent using, for instance, a union-find algorithm.

[8] refines the basic reduction to a decision procedure for a Shostak combination framework in terms of a canonizer and a solver. The solver, still based on a union-find algorithm, is designed to minimize bit-vector decompositions.

3. REDUCING $\mathcal{BV}_C$ TO $\mathcal{E}$

In this section we develop a reduction from $\mathcal{BV}_C$ to $\mathcal{E}$ by means of the key concept of *base*. We show that a base can be used in combination with a union-find algorithm to derive a sound decision procedure for $\mathcal{BV}_C$. In contrast to [8], we reduce bit-vector decompositions by exploiting the notion of base, which elegantly captures the relations between bit-vector slices modulo equality.

3.1 Preliminaries

In the rest of the paper we shall use the term *slice* to refer to an extraction of bits from a bit-vector variable.

DEFINITION 1 (SLICE). *A* proper slice *is any extraction $x_{[n]}[i:j]$, where $x_{[n]}$ is a bit-vector variable, and $0 < j \leq i \leq n-1$ or $0 \leq j \leq i < n-1$. A* slice x^i *of $x_{[n]}$ is either a proper slice or the whole vector of bits, x. We define two slices $x[i_1:j_1]$ and $x[i_2:j_2]$ to be* overlapping *if they share some bits; we say that they are* consecutive *if $j_1 = i_2 + 1$ or $j_2 = i_1 + 1$.*

Notice that a term $p_{[n]} :: q_{[m]} = r$ can be rewritten as $p_{[n]} = r[n+m-1:m], q_{[m]} = r[m-1:0]$, while $p_{[n]}[i_1 : j_1][i_2 : j_2]$ can be rewritten as $p_{[n]}[i_2 + j_1 : j_2 + j_1]$. It can be shown that any equality in $\mathcal{BV}_C$ can be rewritten in terms of *conjunctions of simplified equalities* of the form $p_{[n]} = q_{[n]}$, where p and q are slices. For the sake of explanation, but without loss of expressiveness, from now on we shall assume to deal with conjunctions of simplified equalities.

The main intuition behind our approach is that when reducing $\mathcal{BV}_C$ to $\mathcal{E}$ it is necessary to reason about slices that are *not overlapping*. Intuitively, in any model for a set of $\mathcal{BV}$ constraints, overlapping slices must agree on the values for the bits they share. Notice, however, that a decision procedure for $\mathcal{E}$ applied to a set of $\mathcal{BV}$ constraints considers different slices as independent elements (since in $\mathcal{E}$, core operators are treated as uninterpreted). Consider again E from Example 1. E can be rephrased as a set of equalities involving non-overlapping slices as follows

$$E'' = \left\{ \begin{array}{c} x[5:2] = 0101,\ x[1:0] = 10, \\ y[7:6] = 00,\ y[5:2] = 0110, \\ y[7:6] = x[7:6],\ y[5:2] = x[5:2] \\ y[1:0] = x[1:0] \end{array} \right\}$$

E'' is unsatisfiable in $\mathcal{E}$ for its subset $\{x[5:2] = 0101, x[5:2] = y[5:2], y[5:2] = 0110\}$. The fundamental step in computing E'' from E is choosing a *suitable decomposition* for the variables in E.

DEFINITION 2 (BASE). *Let E be a set of bit-vector equalities. A* base $B(E)$ *is a set of slices of variables in E such that:*

1. *for each variable x, any two slices of x in $B(E)$ are not overlapping*
2. *for each variable x, any slice of x in E is equivalent to* exactly one *concatenation $x^{m-1} :: \ldots :: x^0$, where x^i are consecutive slices of $B(E)$*
3. *it is possible to rewrite each equality $x^i = y^j \in E$ into the form $x^{i_{m-1}} :: \ldots :: x^{i_0} = y^{j_{m-1}} :: \ldots :: y^{j_0}$ where x^{i_k} and y^{j_k} are slices in $B(E)$ having the same width, for $k = 0, \ldots, m-1$*

Intuitively, a base $B(E)$ defines a unique decomposition for a variable x into a concatenation of consecutive slices, in such a way that no two slices share bits, either explicitly or modulo E. A base $B(E)$ can be seen as a function that maps

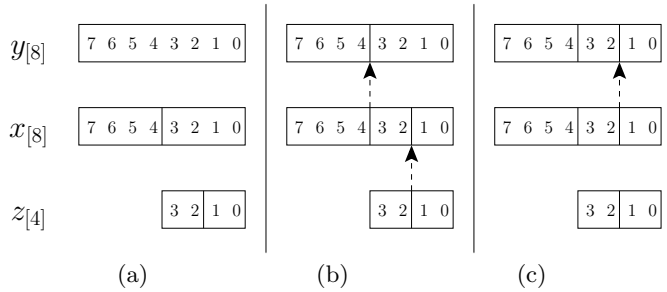

Figure 1: Base computation for Example 2.

each variable x in E to a (unique) decomposition $B(E, x)$. We will omit E from the notation when it is clear from the context. For instance, a base for E in Example 1 is $B = \{x[7:6], x[5:2], x[1:0], y[7:6], y[5:2], y[1:0]\}$, and $B(x) = x[7:6] :: x[5:2] :: x[1:0]$, or simply $B(x) = \{0, 2, 6\}$. We refer to $B(x)$ as to the set of *cut points* of x in the base B. A base can be used to perform a sound reduction from $\mathcal{BV}_C$ to $\mathcal{E}$.

LEMMA 1 ($\mathcal{BV}_C$ REDUCES TO $\mathcal{E}$). *Let $p_{[n]} = q_{[n]}$ be a bit-vector equality, and let B be a base for $p_{[n]} = q_{[n]}$. Let $B(p_{[n]}) = p^m :: \ldots :: p^1$, $B(q_{[n]}) = q^m :: \ldots :: q^1$. We have that $p_{[n]} = q_{[n]}$ is $\mathcal{BV}_C$-satisfiable iff $\bigwedge_{i=1}^{m} p^i = q^i$ is $\mathcal{E}$-satisfiable.*

Notice that a set E may admit more than one base. We define a *partial* order $\preceq$ over the bases of E.

DEFINITION 3 (COARSER BASE). *Let E be a set of bit-vector equalities. Let $B(E)$ and $B'(E)$ be two bases for E. We write $B(E) \preceq B'(E)$ (B is* coarser *than B') iff*

$$\forall x \in Vars(E).\ B(E, x) \subseteq B'(E, x)$$

For instance $B' = \{x[7:6], x[5:4], x[3:2], x[1:0], y[7:7], y[6:6], y[5:2], y[1:0]\}$ is a valid base for E of Example 1.

For any set of equalities E, two special bases are always defined, the *Finest Base* $FB = \{x(i) \mid x_{[n]} \in Vars(E), i = 0, \ldots, n-1\}$, where slices are individual bits, and the *Coarsest Base*, the base CB such that $CB \preceq B$, for any base B. CB (FB) is the base with minimum (maximum) cardinality, that decomposes the variables in E into the largest (smallest) possible chunks.

3.2 Solving $\mathcal{BV}_C$ using the Coarsest Base

The coarsest base for a set of equalities E can be computed with an iterative refinement of a set CB, that keeps track of the cut points for each variable. Each refinement is done by propagating the cut points of the variables according to the equalities in E, until a fix point is reached. We show this process by means of the following example (depicted in Figure 1).

EXAMPLE 2. *Let $E = \{x_{[8]} = y_{[8]},\ x[3:0] = z_{[4]},\ z[3:2] = z[1:0]\}$. Initially $CB = \{x[7:4],\ x[3:0],\ y, z[3:2],\ z[1:0]\}$ (Figure 1a). Since $x = y$ in E, we propagate the cut points of x over to y; since $x[3:0] = z$, we propagate the cut points of z to $x[3:0]$ (Figure 1b). In the second iteration, we propagate again the cut points of x to y, and no further propagation can be done. The resulting coarsest base is $CB = \{x[7:4],\ x[3:2],\ x[1:0],\ y[7:4],\ y[3:2],\ y[1:0],\ z[3:2],\ z[1:0]\}$ (Figure 1c).*

In the following we shall refer to this iterative procedure as to $ComputeCB$[1]. The consistency of a set of $\mathcal{BV}_C$-equalities can be checked by means of a union-find algorithm as shown in Figure 2.

The algorithm that computes the coarsest base ($CheckCB$) works as follows. Initially the coarsest base CB for E is computed (line 1). $SplitEqClasses$ is defined as the procedure that initializes the set of equivalence classes with one class for each element of CB. Equalities in E are processed one at a time (lines 4-6) in the main loop: first, the left- and right-hand side of the equality are rewritten into their decompositions ($CB(p)$, $CB(q)$); equalities among corresponding slices are fed into the union-find procedure ($AssertEq$). $CheckCB$ is correct and complete for $\mathcal{BV}_C$:

THEOREM 1 (CORR. AND COMPL. OF CHECKCB). *Let E be a set of $\mathcal{BV}_C$-equalities. Then E is $\mathcal{BV}_C$-satisfiable iff $CheckCB(E)$ returns true.*

```
function CheckCB(E)
1   CB := ComputeCB(E)
2   SplitEqClasses(CB)
3   foreach p = q ∈ E
4       if ¬AssertBv(CB(p), CB(q))
5           return false
6   return true
end

function AssertBv(p, q)
7   // Let p be p^m :: ... :: p^1
8   // Let q be q^m :: ... :: q^1
9   for i := 1 to m
10      if ¬AssertEq(p^i, q^i)
11          return false
12  return true
end
```

Figure 2: A reduction from $\mathcal{BV}_C$ to $\mathcal{E}$ that uses the coarsest base and relies on a union-find algorithm ($AssertEq$).

3.3 An incremental solver for $\mathcal{BV}_C$

So far we focused on solving a *static* set of $\mathcal{BV}_C$-equalities. In this section we show how to produce an equivalent *incremental* version of the algorithm of §3.2. The incremental version presents two main advantages w.r.t. the static one, (i) it may terminate earlier (ii) it is amenable for integration in a state-of-the-art SMT-Solver. The pseudo-code is shown in Figure 3.

In $CheckCBInc$ equalities are processed one at a time in the main loop of lines 4-9. As opposed to $CheckCB$, this version computes, at any iteration i, the coarsest base for the incrementally growing set of equalities $E^{(i)}$. For two consecutive iterations we have $CB^{(i-1)} \preceq CB^{(i)}$, i.e. the incremental algorithm works with a series of coarsest bases of increasing granularity. This process increases the chances of detecting an inconsistency in E caused by a subset of

[1] $ComputeCB$ always terminates since we are dealing with fixed-width bit-vectors.

```
function CheckCBInc(E)
1   CB^(0) := Vars(E)
2   E^(0) := ∅
3   // Let i be the iteration
4   foreach p = q ∈ E
5      E^(i) := E^(i−1) ∪ {p = q}
6      CB^(i) := ComputeCB(E^(i))
7      SplitEqClasses(CB^(i))
8      if ¬AssertBv(CB^(i)(p), CB^(i)(q))
9         return false
10  return true
end
```

Figure 3: An incremental reduction.

equalities $E^{(i)}$, while reasoning on the coarsest possible base for $E^{(i)}$.

Because of the incremental mechanism, in *CheckCBInc* the coarsest base may change at any iteration; as a result, at some iteration i, $CB^{(i)}$ may contain a slice, say x^i which is not part of $CB^{(i-1)}$. Recall that the introduction of a new equivalence class for x^i would not guarantee the completeness of the algorithm, as x^i may overlap with other slices represented in the union-find algorithm at step i. In fact because objects are uninterpreted in $\mathcal{E}$, the equivalence classes for overlapping slices would be regarded as semantically different objects.

In order to keep the collection of equivalence classes *synchronized* with $CB^{(i)}$ at any iteration i, we call the function *SplitEqClasses* (line 7), which adjusts the status of equivalence classes according to the modification of the coarsest base. It works as follows. When a new slice $x[h:j] \in CB^{(i)}$ needs to be represented, we consider the equivalence class that contains the slice $x[k:l] \in CB^{(i-1)}$, with $k \geq h$ and $l \leq j$, and we replace the equivalence class for $x[k:l]$ with three new equivalence classes for $x[k:h+1]$, $x[h:j]$, and $x[j-1:l]$.

We have the following result:

THEOREM 2 (CORR. AND COMPL. OF CHECKCBINC). *Let E be a set of $\mathcal{BV}_C$-equalities. Then E is $\mathcal{BV}_C$-satisfiable if and only if CheckCBInc(E) returns true.*

EXAMPLE 3. *We run CheckCBInc over E of Example 1. Initially we have $CB^{(0)} = \{x, y\}$ and only two equivalence classes, as shown in Figure 4a. The first equality $x[5:0] =$ 010110 is then considered: $CB^{(1)} = \{x[7:6], x[5:0], y\}$, and the equivalence class for x splits. The equality is passed to AssertBv which updates the equivalence classes. The result is shown in Figure 4b. When $y[7:2] =$ 000110 is considered we have $CB^{(1)} = \{x[7:6], x[5:0], y[7:2], y[1:0]\}$, and the equivalence class for y splits as in Figure 4c. When $x = y$ is processed we have $CB = \{x[7:6],\ x[5:2],\ x[1:0],\ y[7:6],\ y[5:2],\ y[1:0]\}$; the classes for $x[5:0]$ and $y[7:2]$ split as in Figure 4d. The unsatisfiability is detected in the subsequent call to AssertBv.*

4. EFFICIENT BASE REPRESENTATION

In the previous section we presented a decision procedure *CheckCBInc* for $\mathcal{BV}_C$ that splits bit-vectors only when necessary and that relies on a union-find algorithm as a satisfiability procedure. The two key operations in *CheckCBInc* are (i) the incremental computation of the coarsest base and (ii) the splitting of equivalence classes. In this section we show how these two operations can be efficiently implemented *within* the data-structures of a union-find algorithm.

4.1 Representing Decompositions

The decomposition for a variable in terms of consecutive slices can be represented by means of the same data structures used for union-find (recall §2.3) augmented with an auxiliary field `cb`, an ordered list of `(node *)` that points to the nodes representing the sub-slices of the considered bit-vector slice[2]; for each term x, `x->cb` is initialized to `[x]`. When it becomes necessary to represent a new slice of x, say $x[i:j]$, the field `cb` of `x` is updated to

$$\begin{array}{ll} \texttt{[x[n-1:j], x[j-1:0]]} & \text{if } i = n-1 \wedge j > 0 \\ \texttt{[x[n-1:i+1], x[i:0]]} & \text{if } i < n-1 \wedge j = 0 \\ \texttt{[x[n-1:i+1], x[i:j], x[j-1:0]]} & \text{otherwise} \end{array}$$

The idea is to represent the relationships between a bit-vector variable x and its set of its slices as a tree, linked by the `cb` field. Any subsequent slice $x[k:l]$ can be represented with a recursive modification of the slices in the list `x->cb` that represent slices overlapping with $x[k:l]$. The tree generated with this process can be used to retrieve the coarsest decomposition for each slice.

EXAMPLE 4. *Suppose that we need representing, in order, two slices, $x[5:0]$ and $x[7:3]$, for a bit-vector $x_{[8]}$. Initially `x->cb` is `[x]` (Figure 5a). In order to represent $x[5:0]$, we update `x->cb` to `[ x[7:6],x[5:0] ]` (Figure 5b). When considering $x[7:3]$ we recursively update the nodes in `x->cb` that overlap with $x[7:3]$, i.e., we update `x[5:0]->cb` to `[ x[5:3], x[2:0] ]` as in Figure 5c. The leaves of the tree rooted in `x` and linked by the `cb` fields is the coarsest decomposition for x.*

4.2 Representing and Manipulating the Coarsest Base

It is easy to extend the representation for the coarsest decompositions to a representation for the coarsest base: it is sufficient to apply the aforementioned transformations *modulo equivalence classes*. Precisely, whenever we need to update the coarsest decomposition for a slice p, we recursively update the *representants* of the slices linked by the field `p->cb`. The data structure resulting from this process is a DAG, whose leaves are, at each iteration, the representants of the equivalence classes defining the coarsest base. We call this representation *coarsest base modulo equivalence*, *CBE* for short.

When we modify the *CBE* with the addition of a new slice, we obtain at the same time the computation of a new coarsest base and the splitting of equivalence classes (lines 8-9 of Figure 3), as the coarsest base is implicitly represented in the set of equivalence classes: a leaf `p` in the *CBE* represents the equivalence class of the slices of the terms that can reach `p` by traversing the *CBE*.

EXAMPLE 5. *We show a possible evolution of the CBE for E of Example 2. Initially $CB^{(0)} = \{x,\ y,\ z\}$. When processing $x = y$ we adjust `y->rep` to point to `x` . When processing $x[3:0] = z$ we allocate `x[7:4]` and `x[3:0]`, we update `x->cb` to `[x[7:4],x[3:0]]`, and we set `x[3:0]->rep`*

[2]In our approach only three pointers to `node` are sufficient. We use a list for simplicity in the description.

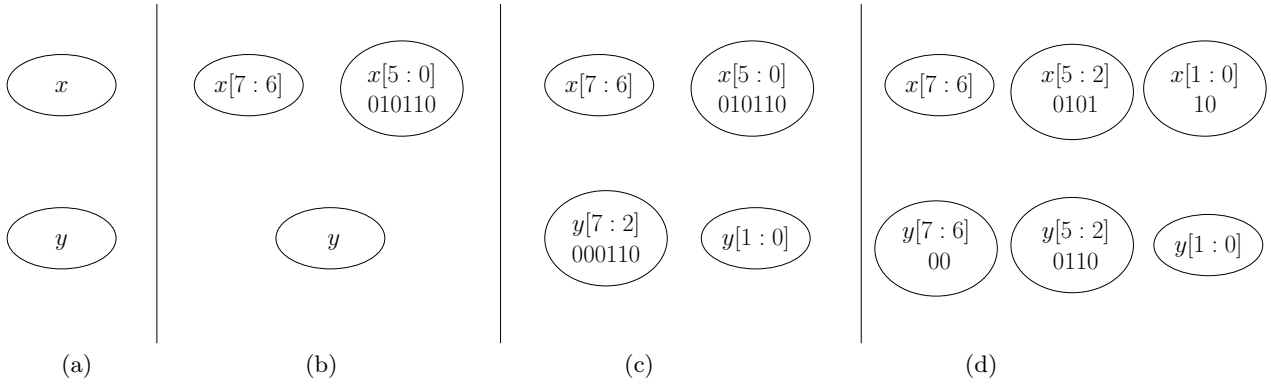

Figure 4: Equivalence classes maintained by *CheckCBInc* in Example 3

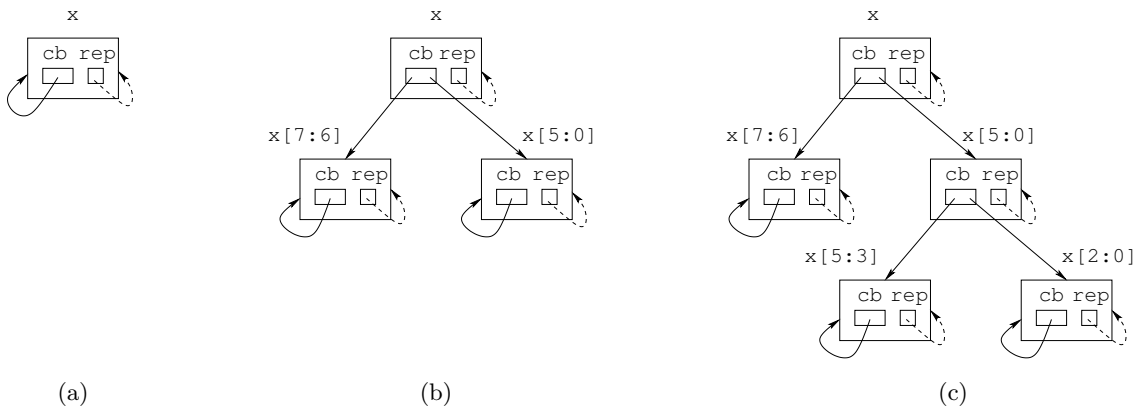

Figure 5: Coarsest decompositions for x in Example 4: (a) initial, (b) after representing $x[5:0]$, (c) after representing $x[7:3]$.

to z*, as in Figure 6a. When processing* $z[3:2] = z[1:0]$ *we allocate* z[3:2] *and* z[1:0] *and adjust the pointers as in Figure 6b. The leaves of the final CBE,* x[7:4] *and* z[1:0]*, represent, respectively, the two equivalence classes* $\{x[7:4], y[7:4]\}$ *and* $\{x[3:2], x[1:0], y[3:2], y[1:0], z[3:2], z[1:0]\}$.

5. EXTENSIONS FOR LAZY SMT

For the sake of simplicity, we described *CheckCBInc* in terms of manipulations of variables and extractions over variables by means of a union-find algorithm. However the idea can be straightforwardly extended to the case of generic $\mathcal{BV}$ terms handled by a congruence closure algorithm; it is sufficient to keep the CBE for any term whose outmost operator is not extraction. For example the set $\{(x+y)[5:0] = 0001, (x+y)[7:3] = 00100\}$ can be proven inconsistent using our algorithm, by applying the same reasoning on the uninterpreted term $(x+y)$. In other words, our algorithms improves the congruence closure algorithm by giving *explicit interpretation* to the extraction operation.

The new data structure is built on top of the congruence closure algorithm of [10], thus inheriting the incremental and backtrackable mechanism, and the ability of handling negated equalities[3] (using the same infrastructure described in [10]). For computing explanations we adapted the proof-production method of [17]. Finally notice that our procedure can be used as a theory-layer before resorting to other complete decision procedures for $\mathcal{BV}$.

[3]We do not check finite-domain conditions, though.

6. EXPERIMENTS

We evaluated the decision procedure based on CBE with respect to bit-blasting. We implemented both algorithms inside OpenSMT [19]. The bit-blaster uses a state-of-the-art SAT-Solver (MiniSAT 2).

We compare two versions of OpenSMT: baseline implements a standard congruence closure algorithm plus bit-blasting, while cbe implements the congruence closure modified with CBE plus bit-blasting. In both versions, bit-blasting is called only if the congruence closure fails to detect an inconsistency, and only when a complete model is enumerated.

In order to test the effectiveness of CBE we used three suites of crafted benchmarks. The benchmarks are suitable to test CBE as they stimulate (i) its incremental and backtrackable behavior (requiring rich disjuntions of constraints), and (ii) the mechanism of handling multiple bit-vector slices (requiring core operators that could not be trivially simplified away). The benchmarks, included in the SMT-LIB, are described as follows:

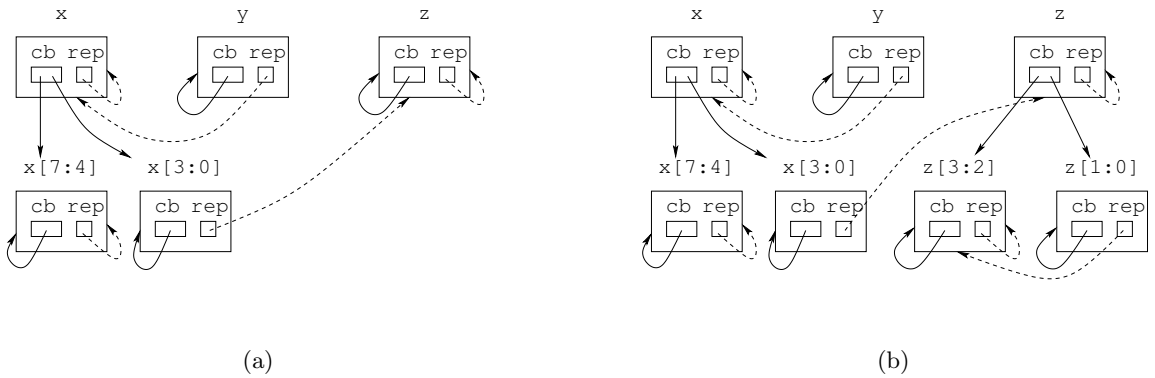

Figure 6: Evolution of CBE for Example 5.

ec: encodes a set of basic properties on $\mathcal{BV}_C$, such as $(x[15:4] = y[15:8] :: z \wedge x[11:0] = z :: y[7:0]) \rightarrow y[11:8] = y[7:4]$. Benchmarks are parametric in the number and in the width of the variables, and in the width of the smallest slice in the formula. Since CBE is sufficient to decide these benchmarks, with this test we evaluate the *effectiveness of CBE with respect to bit-blasting.*

sp: encodes the behavior of n simple processors, based on the Verilog code of Example 11.2 of [18]. Benchmarks are parametric in the number of processors and in the width of the data bus. They are defined over $\mathcal{BV}$, but some theory-conflicts are in $\mathcal{BV}_C$. With these benchmarks we evaluate the *effectiveness of CBE used as a theory-layer.*

lfsr: encodes the behavior of a linear feedback shift register, a circuit commonly used in pseudo-random number generation. The benchmarks are parametric in the number and width of registers, and the number of clock tics. We verify that a lfsr cycles over non-zero values, unless the reset pin is activated. In these benchmarks no conflict is due to $\mathcal{BV}_C$ atoms (every call to CBE is satisfiable). With these benchmarks we evaluate the overhead of the CBE layer. It is an *empirical evaluation of the computational cost* of our procedure.

As a reference, we include in the comparison the best four solvers that participated in the SMTCOMP'08. As far as we know, they are mostly based on bit-blasting. Notice, however, that each of them implements a number of different algorithms for preprocessing, and optimizations for the bit-blaster, which makes it difficult to judge a direct comparison with our solver. Nevertheless we shall provide data for the other solvers as an indication of the complexity of the benchmarks used in our evaluation. Table 1 shows the number of instances that each solver was able to process within a timeout of 1800 s, on an Intel Xeon 3.4 GHz running Linux. Scatter-plots in Figure 7-9 detail the comparison between the two variants of OpenSMT.

	OpenSMT		State-of-the-art Solvers			
Suite	cbe	bsln	Boolector v0.4	Z3 v3.2	Beaver v1.0	MathSAT v4.2
ec	654	651	136	322	644	608
sp	37	12	32	44	39	28
lfsr	188	190	146	228	221	231

Table 1: Number of instances solved by each solver within 1800 s.

Experiments show that using CBE stand-alone over $\mathcal{BV}_C$ (ec) results in a better performance than using bit-blasting (see also Figure 7). In this case OpenSMT is also superior to the other solvers. Using CBE as a theory-layer over $\mathcal{BV}$ (sp) gives a considerable speedup over plain bit-blasting: the activation of the CBE layer makes OpenSMT comparable to the other SMT-Solvers. The last row of Table 1 (lfsr) shows that the reasoning on CBE data structures can be done quite efficiently: it does not produce a significant overhead over the baseline version when the reasoning on $\mathcal{BV}_C$ is not necessary.

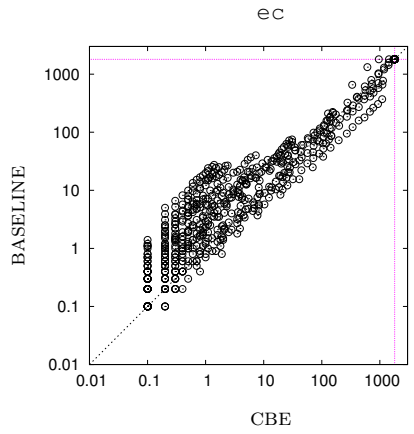

Figure 7: Scatter-plots comparing the two variants of OpenSMT, with the CBE layer active (cbe) and disabled (baseline).

7. CONCLUSION

We presented a new efficient decision procedure for the core theory of bit-vectors that relies on a reduction to the theory of equality $\mathcal{E}$. We formalized the reduction by introducing the notion of *base*, a decomposition for bit-vectors that ensures the completeness of the reduction. In order to efficiently represent the base, we introduced a novel data-structure, the CBE, a DAG that stores the decompositions for terms modulo the equivalence classes. One of the advantages of the CBE is that it can be implemented with a minor modification of the similar data-structures of a state-of-the-art congruence closure algorithm.

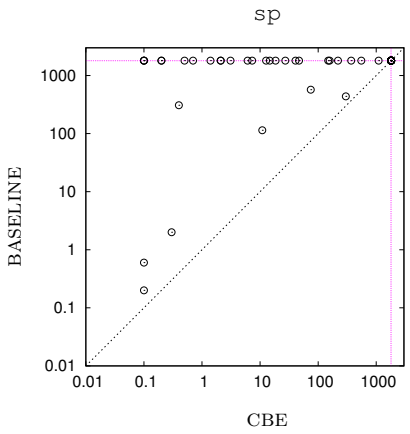

Figure 8: Scatter-plots comparing the two variants of OpenSMT, with the *CBE* layer active (CBE) and disabled (BASELINE).

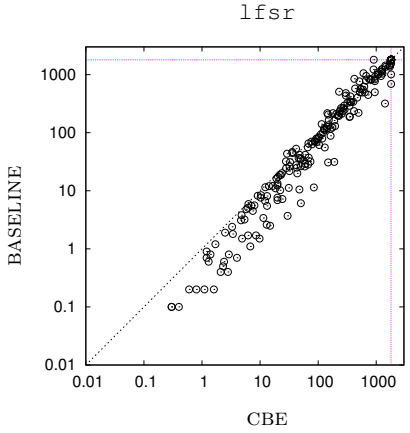

Figure 9: Scatter-plots comparing the two variants of OpenSMT, with the *CBE* layer active (CBE) and disabled (BASELINE).

Because the *CBE* can be easily modified or restored to a previous state, the procedure supports incrementality and efficient backtracking. A partial support for negated equalities and uninterpreted functional symbols is automatically inherited from the congruence closure algorithm. We implemented the new procedure in our tool, OpenSMT. Our experimentation and comparison with the state-of-the-art SMT solvers show that the new approach is comparable and often superior to bit-blasting on the core operations, and furthermore, it also improves efficiency when applied over the full bit-vector theory.

8. REFERENCES

[1] C. Barrett and C. Tinelli. CVC3. In *CAV'07*, 2007.

[2] C. W. Barrett, D. L. Dill, and J. R. Levitt. A Decision Procedure for Bit-Vector Arithmetic. In *DAC*, pages 522–527, 1998.

[3] A. Biere and R. Brummayer. The Boolector SMT Solver. In *TACAS*, 2009.

[4] N. Bjørner and M. C. Pichora. Deciding Fixed and Non-fixed Size Bit-vectors. In *TACAS*, pages 376–392, 1998.

[5] R. Bruttomesso, A. Cimatti, A. Franzen, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani. A Lazy and Layered SMT(BV) Solver for Hard Industrial Verification Problems. In *CAV*, pages 247–260, 2007.

[6] Roberto Bruttomesso. *RTL Verification: from SAT to SMT(BV)*. PhD thesis, University of Trento, 2008. Available at `http://www.inf.unisi.ch/postdoc/bruttomesso/files/phdthesis.pdf`.

[7] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In *CAV*, 2002.

[8] D. Cyrluk, M. O. Möller, and H. Rueß. An Efficient Decision Procedure for the Theory of Fixed-Sized Bit-Vectors. In *CAV*, pages 60–71, 1997.

[9] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS'08*, pages 337–340, 2008.

[10] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of ACM*, 52(3):365–473, 2005.

[11] B. Dutertre and L. de Moura. The Yices SMT Solver. Tool paper available at `http://yices.csl.sri.com/tool-paper.pdf`.

[12] V. Ganesh, S. Berezin, and D. L. Dill. A Decision Procedure for Fixed-Width Bit-Vectors. Technical report, University of Stanford, 2005. Available at `http://theory.stanford.edu/~vganesh/bitvector-tech-report.ps`.

[13] V. Ganesh and D. L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *CAV*, pages 519–531, 2007.

[14] S. Jha, R. Limaye, and S. A. Seshia. Beaver: Engineering an Efficient SMT Solver for Bit-Vector Arithmetic. In *CAV*, 2009.

[15] D. E. Knuth and A. Schönhage. The expected linearity of a simple equivalence algorithm. *Theoretical Computer Science*, 3(6):281–315, June 1978.

[16] P. Manolios, S. K. Srinivasan, and D. Vroon. BAT: The Bit-Level Analysis Tool. In *CAV*, pages 303–306, 2007.

[17] R. Nieuwenhuis and A. Oliveras. Proof-Producing Congruence Closure. In *RTA'05*, pages 453–468, 2005.

[18] Peter M. Nyasulu. Introduction to Verilog. Available at `http://www.doe.carleton.ca/~shams/97350/PetervrlK.pdf`.

[19] OpenSMT. `http://verify.inf.unisi.ch/opensmt`.

[20] R. Sebastiani. Lazy Satisfiability Modulo Theories. *JSAT*, 3:144–224, 2007.

[21] Spear. `http://www.cs.ubc.ca/~babic/index_spear.htm`.

[22] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

[23] R. Wille, G. Fey, D. Große, S. Eggersglüß, and R. Drechsler. SWORD: A SAT like prover using word level information. In *VLSI-SoC*, pages 88–93. IEEE, 2007.